IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Patent No.	:	**6,308,317**
Application No.	:	**08/957,512**
Applicant	:	**Wilkinson, Timothy J., et al.**
Issue Date	:	**Oct. 23, 2001**
Docket No.	:	**40.0010**
Customer No.	:	**41754**

CERTIFICATE OF ELECTRONIC TRANSMISSION UNDER 37 CFR 1.8

Date of Transmission: July 30, 2010

I hereby certify that this correspondence is being filed with the United States Patent and Trademark Office using the Electronic Filing System on the date indicated above.

/Pehr Jansson/
Pehr Jansson, Reg. No. 35,759

Certificate of Correction Branch
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Petition for Correction of an Applicant Mistake

Dear Sir:

Applicant hereby petitions for acceptance of the herewith-filed Certificate of Correction to correct a typographical mistake. U.S. Patent No. 6,308,317 (the '317 patent) issued October 23, 2001. It was subsequently reexamined (90/008,178), and a Reexamination Certificate was issued on December 2, 2008. During the reexamination process Applicant modified several claims and made a typographical error. Applicant petitions to have this error corrected.

Facts

The '317 patent relates to the use of a high level programming language, such as Java, in a resource-constrained environment. The specification discloses and claims the invention in the context of two resource-constrained environments or devices, including for an "integrated circuit card" and a "microcontroller." Nearly verbatim claim limitations are recited in the claims with some claims reciting an "integrated circuit card" in the preamble to the invention and others reciting a "microcontroller" instead (*compare, e.g.*, claims 1 and 58). During reexamination, Applicant modified claim 1, directed to an integrated circuit card, by amending the limitation "in a format suitable for interpretation" to recite "in an instruction set supported by an interpreter on the *integrated circuit card*." 90/008,178, **REPLY AND AMENDMENT UNDER 37 CFR 1.111 and 37 CFR 1.530**, filed February 21, 2008 (Attached hereto as Exhibit A). Applicant made this amendment a global amendment to all occurrences of the same limitation in the claims, and, by doing so, accidentally neglected the fact that there are some claims to the invention that recite a "microcontroller" as the resource-constrained device. Because these claims recite a "microcontroller," not an "integrated circuit card," the amendment ("in an instruction set supported by an interpreter on the *integrated circuit card*") erroneously refers back to "the integrated circuit card" when it is clear (because the resource-constrained device recited in the preamble is a microcontroller) it should refer back to a "*microcontroller*." Applicant thus petitions to replace "integrated circuit card" in claims 58, 65, 78, 87, 88, and 92 (all of which recite a microcontroller in the preamble) with "microcontroller" to correct this typographical error.

Authority

Under certain circumstances, an Applicant mistake during prosecution of a patent may be corrected by Certificate of Correction. 35 U.S.C. § 255; *see*

also 37 CFR 1.323 and MPEP 1481. Specifically, if the following conditions are satisfied, a Certificate of Correction may be used to correct an Applicant mistake:

> (A) mistake must be of a clerical or typographical nature, or of minor character, the correction of which does not involve such changes in the patent as would constitute new matter or would require reexamination;
>
> (B) which was not the fault of the Patent and Trademark Office;
>
> (C) such mistake occurred in good faith; and
>
> (D) payment of the required fee.

35 U.S.C. § 255. In the case of the '317 patent, all of these requirements are satisfied. Accordingly, this Petition for Correction of an Applicant Mistake should be granted.

Regarding element (A), the Federal Circuit has established that a Certificate of Correction may be used to correct clear typographical errors, including those in the claims. In *Arthocare v. Smith & Nephew*, the Federal Circuit evaluated a Certificate of Correction to correct an Applicant mistake and held, "The correction of a ministerial error in the claims, which also serves to broaden the claims, is allowable if it is 'clearly evident from the specifications, drawings, and prosecution history how the error should appropriately be corrected' to one of skill in the art." 406 F.3d 1365, 1374-75 (Fed. Cir. 2005) (citing *Superior Fireplace Co. v. Majestic Prods. Co.*, 270 F.3d 1358, 1373 (Fed. Cir. 2001)). In *Arthocare*, Applicant meant to globally change all occurrences of the term "active electrode" in the claims to "electrode terminal," but accidentally left a few occurrences unchanged. *Id.* at 1374 ("Those amendments changed the term 'active electrode' to 'electrode terminal' in three places in claim 1 … but did not make the change in a fourth place, where the term 'active electrode' was left unchanged.") Claims that erroneously contained both terms appeared to call for an additional type

of electrode. *Id.* at 1375. Applicant realized his mistake after the patent issued and filed for and received a Certificate of Correction to replace the remaining occurrences of "active electrode" with the term "electrode terminal," which the PTO issued. *Id.* at 1374. The Federal Circuit held the Certificate of Correction valid.

The Federal Circuit reviewed the specification and prosecution history and found it clear that the prosecuting attorney made a "typographical error" and that it was proper to change "active electrode" to "electrode terminal" using a Certificate of Correction, even though the correction had the affect of broadening the claim. *Id.* at 75 ("In the first place, claim 1 does not make sense if it is interpreted to contain three types of electrodes instead of two.... The prosecution history further supports ArthroCare's argument that is was unambiguous how the remaining reference to an active electrode in claim 1 should be changed.... The prosecuting attorney's failure to replace the term "active electrode' twice in the claims instead of once, does not demonstrate ... that a person of ordinary skill in the art would not understand how to correct those errors.") Thus, a Certificate of Correction is an appropriate means to change a clear typographical error in the claims, if it is clearly evident how to correct the error.

Analysis

Please consider these conditions one-by-one in order.

(A) mistake must be of a clerical or typographical nature, or of minor character, the correction of which does not involve such changes in the patent as would constitute new matter or would require reexamination;	The specification and prosecution history make clear that Applicant made a typographical error. The amendment refers to "the integrated circuit card," which is a reference back to a prior use of "integrated circuit card" in the claims. The term "integrated circuit card" does not appear in the microcontroller claims, so there is no antecedent basis for use of "the integrated circuit card." The error occurred when the prosecuting attorney crafted the amendment for claim 1, which is directed to an integrated circuit card, and then decided to use this amendment globally, neglecting the fact that some independent claims with the same limitation are directed to a microcontroller. How to correct the error is clearly evident, including to one of ordinary skill in the art, as demonstrated by the attached Certificate of Correction.
(B) which was not the fault of the Patent and Trademark Office;	The mistake was made by Applicant in the February 21, 2008 Amendments to the claims and related Remarks. Thus, the mistake is not the fault of the Patent and Trademark Office.
(C) such mistake occurred in good faith; and	As noted above, the specification and prosecution history make clear that Applicant made a mistake when amending the microcontroller claims. The undersigned of this Petition hereby swears that he is the same attorney that made the amendment during the '317 reexamination and that the mistake was in good faith.
(D) payment of the required fee.	The fee set forth in MPEP § 1.20(a) is included with this Petition.

Thus, all the requirements of for accepting a Certificate of Correction to correct Applicant's mistake under 35 U.S.C §355 apply here. Accordingly, Patentee petitions the Commissioner to accept the attached Certificate of Correction.

Respectfully submitted,

Date: July 30, 2010

_____/Pehr Jansson/_____
Pehr Jansson
Registration No. 35,759

The Jansson Firm
9501 N. Capital of TX Hwy. #202
Austin, TX 78759
512-372-8440
512-597-0639 (Fax)
pehr@thejanssonfirm.com

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Application No.	:	90/008,178
Patent No.	:	6,308,317
Issue Date	:	October 23, 2001
Patentee	:	Axalto Inc.
Inventors	:	TIMOTHY J. WILKINSON, SCOTT B. GUTHERY, KSHEERABDHI KRISHNA, MICHAEL A MONTGOMERY
For	:	USING A HIGH LEVEL PROGRAMMING LANGUAGE WITH A MICROCONTROLLER
Confirmation No.	:	9953

Mail Stop "Ex Parte Reexam" Central Reexamination Unit
Commissioner for Patents
P.O.Box 1450
Alexandria, VA, 22313-1450

REPLY AND AMENDMENT UNDER 37 CFR 1.111 and 37 CFR 1.530

Dear Sir:

In response to the Office Action of 21 December 2007, with a period for to expire after February 21, 2008, please amend the above-identified application as follows.

Amendments to the Claims are reflected in the listing of claims which begins on page 2 of this paper.

Remarks/Arguments begin on page 30 of this paper.

An Appendix including quoted pages from Zhiqun Chen, Java Card ™ Technology for Smart Cards, Sun Microsystems (2000) is attached following page 72.

Each section begins on a separate sheet in accordance with the revised format practice.

Amendments to the claims:

1. (Amended) An integrated circuit card for use with a terminal, comprising:

a communicator configured to communicate with the terminal;

a memory storing:

an application derived from a program written in a high level programming language format wherein the application is derived from a program written in a high level programming language format by first compiling the program into a compiled form and then converting the compiled form into a converted form, the converting step [including at least one step selected from a group consisting of] comprising:

recording all jumps and their destinations in the original byte codes;

performing a conversion operation selected from the group:

converting specific byte codes into equivalent generic byte codes[or vice-versa];

modifying byte code operands from references using identifying strings to references using unique identifiers; and

renumbering byte codes in [a compiled format] the compiled form to equivalent byte codes in an instruction set supported by an interpreter on the integrated circuit card[a format suitable for interpretation]; and

relinking jumps for which the destination address is affected by the conversion operation; and

an interpreter operable to interpret such an application derived from a program written in a high level programming language format; and

a processor coupled to the memory, the processor configured to use the interpreter to interpret the application for execution and to use the communicator to communicate with the terminal.

2. The integrated circuit card of claim 1, wherein the high level programming language format comprises a class file format.

3. The integrated circuit card of claim 1 wherein the processor comprises a microcontroller.

4. The integrated circuit card of claim 1 wherein at least a portion of the memory is located in the processor.

5. The integrated circuit card of claim 1 wherein the high level programming language format comprises a Java programming language format.

6. The integrated circuit card of claim 1, wherein

the application has been processed from a second application having a plurality of program elements, at least one being a string of characters, and

wherein in the first application the string of characters is replaced with an identifier.

7. The integrated circuit card of claim 6, wherein the identifier comprises an integer.

8. The integrated circuit card of claim 1 wherein the processor is further configured to:

receive a request from a requester to access an element of the card;

after receipt of the request, interact with the requester to authenticate an identity of the requester; and

based on the identity, selectively grant access to the element.

9. The integrated circuit card of claim 8, wherein the requester comprises the processor.

10. The integrated circuit card of claim 8, wherein the requester comprises the terminal.

11. The integrated circuit card of claim 8, wherein

the element comprises the application stored in the memory, and

once access is allowed, the requester is configured to use the application.

12. The integrated circuit card of claim 8, wherein

the element comprises another application stored in the memory.

13. The integrated circuit card of claim 8, wherein the element includes data stored in the memory.

14. The integrated circuit card of claim 8 wherein the element comprises the communicator.

15. The integrated circuit card of claim 8, wherein the memory also stores an access control list for the element, the access control list furnishing an indication of types of access to be granted to the identity, the processor further configured to:

based on the access control list, selectively grant specific types of access to the requester.

16. The integrated circuit card of claim 15 wherein the types of access include reading data.

17. The integrated circuit card of claim 15 wherein the types of access include writing data.

18. The integrated circuit card of claim 15 wherein the types of access include appending data.

19. The integrated circuit card of claim 15 wherein the types of access include creating data.

20. The integrated circuit card of claim 15 wherein the types of access include deleting data.

21. The integrated circuit card of claim 15 wherein the types of access include executing an application.

22. The integrated circuit card of claim 1, wherein the application is one of a plurality of applications stored in the memory, the processor is further configured to:

receive a request from a requester to access one of the plurality of applications;

after receipt of the request, determine whether said one of the plurality of applications complies with a predetermined set of rules; and

based on the determination, selectively grant access to the requester to said one of the plurality of applications.

23. The integrated circuit card of claim 22, wherein the predetermined rules provide a guide for determining whether said one of the plurality of applications accesses a predetermined region of the memory.

24. The integrated circuit card of claim 22, wherein the processor is further configured to:

authenticate an identity of the requester; and

grant access to said one of the plurality of applications based on the identity.

25. The integrated circuit card of claim 1, wherein the processor is further configured to:

interact with the terminal via the communicator to authenticate an identity; and

determine if the identity has been authenticated; and

based on the determination, selectively allow communication between the terminal and the integrated circuit card.

26. The integrated circuit card of claim 25, wherein the communicator and the terminal communicate via communication channels, the processor further configured to assign one of the communication channels to the identity when the processor allows the communication between the terminal and the integrated circuit card.

27. The integrated circuit card of claim 26, wherein the processor is further configured to:

assign a session key to said one of the communication channels, and

use the session key when the processor and the terminal communicate via said one of the communication channels.

28. The integrated circuit card of claim 1, wherein the terminal has a card reader and the communicator comprises a contact for communicating with the card reader.

29. The integrated circuit card of claim 1, wherein the terminal has a wireless communication device and the communicator a wireless transceiver for communicating with the wireless communication device.

30. The integrated circuit card of claim 1, wherein the terminal has a wireless communication device and the communicator comprises a wireless transmitter for communicating with the wireless communication device.

31. (Amended) A method for use with an integrated circuit card and a terminal, comprising:

storing an interpreter operable to interpret programs derived from programs written in a high level programming language and an application derived from a program written in a high level programming language format in a memory of the integrated circuit card wherein the application is derived from a program written in a high level programming language format by first compiling the program into a compiled form and then converting the compiled form into a converted form, the converting step comprising: [including at least one step selected from a group consisting of]

recording all jumps and their destinations in the original byte codes;

performing a conversion operation selected from the group:

converting specific byte codes into equivalent generic byte codes[or vice-versa];

modifying byte code operands from references using identifying strings to references using unique identifiers; and

renumbering byte codes in [a compiled format] the compiled form to equivalent byte codes in an instruction set supported by an interpreter on the integrated circuit card[a format suitable for interpretation]; and

relinking jumps for which the destination address is affected by the conversion operation; and

using a processor of the integrated circuit card to use the interpreter to interpret the application for execution; and

using a communicator of the card when communicating between the processor and the terminal.

32. The method of claim 31, wherein the high level programming language format comprises a class file format.

33. The method of claim 31, wherein the processor comprises a microcontroller.

34. The method of claim 31, wherein at least a portion of the memory is located in the processor.

35. The method of claim 31, wherein the high level programming language format comprises a Java programming language format.

36. The method of claim 31, wherein

the application has been processed from a second application having a plurality of program elements, at least one being a string of characters, further comprising:

replacing the string of characters in the first application with an identifier.

37. The method of claim 36, wherein the identifier includes an integer.

38. The method of claim 31, further comprising:

receiving a request from a requester to access an element of the card;

after receipt of the request, interacting with the requester to authenticate an identity of the requester; and

based on the identity, selectively granting access to the element.

39. The method of claim 38, wherein the requester comprises the processor.

40. The method of claim 38, wherein the requester comprises the terminal.

41. The method of claim 38, wherein the element comprises the application stored in the memory, further comprising:

once access is allowed, using the application with the requester.

42. The method of claim 38, wherein the element comprises another application stored in the memory.

43. The method of claim 38, wherein the element includes data stored in the memory.

44. The method of claim 38, wherein the element comprises the communicator.

45. The method of claim 38, wherein the memory also stores an access control list for the element, the access control list furnishing an indication of types of access to be granted to the identity, further comprising:

based on the access control list, using the processor to selectively grant specific types of access to the requester.

46. The method of claim 45, wherein the types of access include reading data.

47. The method of claim 45, wherein the types of access include writing data.

48. The method of claim 45, wherein the types of access include appending data.

49. The method of claim 45, wherein the types of access include creating data.

50. The method of claim 45, wherein the types of access include deleting data.

51. The method of claim 45, wherein the types of access include executing an application.

52. The method of claim 31, wherein the application is one of a plurality of applications stored in the memory, further comprising:

receiving a request from a requester to access one of the applications stored in the memory;

upon receipt of the request, determining whether said one of the plurality of applications complies with a predetermined set of rules; and

based on the determining, selectively granting access to the said one of the plurality of applications.

53. The method of claim 52, wherein the predetermined rules provide a guide for determining whether said one of the plurality of applications accesses a predetermined region of the memory.

54. The method of claim 52, further comprising:

authenticating an identity of the requester; and

based on the identity, granting access to said one of the plurality of applications.

55. The method of claim 31, further comprising:

communicating with the terminal to authenticate an identity;

determining if the identity has been authenticated; and

based on the determining, selectively allowing communication between the terminal and the integrated circuit card.

56. The method of claim 55, further comprising:

communicating between the terminal and the processor via communication channels; and

assigning one of the communication channels to the identity when the allowing allows communication between the card reader and the integrated circuit card.

57. The method of claim 56, further comprising:

assigning a session key to said one of the communication channels; and

using the session key when the processor and the terminal communicate via said one of the communication channels.

58. (Amended) A microcontroller comprising:

a memory storing:

a derivative application derived from an application having a class file format wherein the application is derived from an application having a class file format by first compiling the application having a class file format into a compiled form and then converting the compiled form into a converted form, the converting step comprising:[including at least one step selected from a group consisting of]

recording all jumps and their destinations in the original byte codes;

performing a conversion operation selected from the group:

converting specific byte codes into equivalent generic byte codes[or vice-versa];

modifying byte code operands from references using identifying strings to references using unique identifiers; and

renumbering byte codes in [a compiled format] the compiled form to equivalent byte codes in an instruction set supported by an interpreter on the integrated circuit card[a format suitable for interpretation], and

relinking jumps for which the destination address is affected by the conversion operation; and

an interpreter configured to interpret applications derived from applications having a class file format; and

a processor coupled to the memory, the processor configured to use the interpreter to interpret the derivative application for execution.

59. The microcontroller of claim 58, further comprising:
a communicator configured to communicate with a terminal.

60. The microcontroller of claim 59, wherein the terminal has a card reader and the communicator comprises a contact for communicating with the card reader.

61. The microcontroller of claim 59, wherein the terminal has a wireless communicator and a wireless transceiver for communicating with the wireless communication device.

62. The microcontroller of claim 59, wherein the terminal has a wireless communication device and the communicator comprises a wireless transmitter for communicating with the wireless communication device.

63. The microcontroller of claim 58, wherein the class file format comprises a Java class file format.

64. (Amended) An integrated circuit card for use with a terminal, comprising:

a communicator configured to communicate with the terminal;

a memory storing:

applications, each application derived from applications having a high level programming language format, and

an interpreter operable to interpret applications derived from applications having a high level programming language format wherein [the]

each application is derived from a program written in a high level programming language format by first compiling the program into a compiled form and then converting the compiled form into a converted form, the converting step[including at least one step selected from a group consisting of]comprising:

recording all jumps and their destinations in the original byte codes;

performing a conversion operation selected from the group:

converting specific byte codes into equivalent generic byte codes[or vice-versa];

modifying byte code operands from references using identifying strings to references using unique identifiers; and

renumbering byte codes in [a compiled format] the compiled form to equivalent byte codes in an instruction set supported by an interpreter on the integrated circuit card[a format suitable for interpretation]; and

relinking jumps for which the destination address is affected by the conversion operation; and

a processor coupled to the memory, the processor configured to:

a.) use the interpreter to interpret the applications for execution,

b.) use the interpreter to create a firewall to isolate the applications from each other, and

c.) use the communicator to communicate with the terminal.

65. (Amended) A microcontroller having a set of resource constraints and comprising:

a memory, and

an interpreter loaded in memory and operable within the set of resource constraints,

the microcontroller having: at least one application loaded in the memory to be interpreted by the interpreter, wherein the at least one application is generated by a programming environment comprising:

a) a compiler for compiling application source programs written in high level language source code form into a compiled form, and

b) a converter for post processing the compiled form into a minimized form suitable for interpretation within the set of resource constraints by the interpreter, wherein the converter comprises means for translating from the byte codes in the compiled form to byte codes in a format suitable for interpretation by the interpreter by:

[a) using at least one step in a process including the steps:

a]b.1) recording all jumps and their destinations in the original byte codes;

b.2) performing a conversion operation selected from the group:

[a.2]b.2.1) converting specific byte codes into equivalent generic byte codes[or vice-versa];

[a.3]b.2.2) modifying byte code operands from references using identifying strings to references using unique identifiers; and

[a.4]b.2.3) renumbering byte codes in [the compiled format] the compiled form to equivalent byte codes in an instruction set supported by an interpreter on the integrated circuit card[a format suitable for interpretation]; and

b.3) relinking jumps for which the destination address is [effected] affected by the conversion operation[step a.1, a.2, a.3, or a.4].

66. The microcontroller of Claim 65, wherein the compiled form includes attributes, and the converter comprises a means for including attributes required by the interpreter while not including the attributes not required by the interpreter.

67. The microcontroller of Claim 65 wherein the compiled form is in a standard Java class file format and the converter accepts as input the compiled form in the standard Java class file format and produces output in a form suitable for interpretation by the interpreter.

68. The microcontroller of Claim 65 wherein the compiled form includes associating an identifying string for objects, classes, fields, or methods, and the converter comprises a means for mapping such strings to unique identifiers.

69. The microcontroller of Claim 68 wherein each unique identifier is an integer.

70. The microcontroller of Claim 68 wherein the mapping of strings to unique identifiers is stored in a string to identifier map file.

71. The microcontroller of Claim 65 where in the high level language supports a first set of features and a first set of data types and the interpreter supports a subset of the first set of features and a subset of the first set of data types, and wherein the converter verifies that the compiled form only

contains features in the subset of the first set of features and only contains data types in the subset of the first set of data types.

72. The microcontroller of Claim 65 wherein the application program is compiled into a compiled form for which resources required to execute or interpret the compiled form exceed those available on the microcontroller.

73. The microcontroller of Claim 65 wherein the compiled form is designed for portability on different computer platforms.

74. The microcontroller of Claim 65 wherein the interpreter is further configured to determine, during an interpretation of an application, whether the application meets a security criteria selected from a set of rules containing at least one rule selected from the set:

not allowing the application access to unauthorized portions of memory,

not allowing the application access to unauthorized microcontroller resources,

wherein the application is composed of byte codes and checking a plurality of byte codes at least once prior to execution to verify that execution of the byte codes does not violate a security constraint.

75. The microcontroller of Claim 65 wherein at least one application program is generated by a process including the steps of:

prior to loading the application verifying that the application does not violate any security constraints; and

loading the application in a secure manner.

76. The microcontroller of Claim 75 wherein the step of loading in a secure manner comprises the step of:

verifying that the loading identity has permission to load applications onto the microcontroller.

77. The microcontroller of Claim 75 wherein the step of loading in a secure manner comprises the step of:

encrypting the application to be loaded using a loading key.

78. (Amended) A method of programming a microcontroller having a memory and a processor operating according to a set of resource constraints, the method comprising the steps of:

inputting an application program in a first programming language;

compiling the application program in the first programming language into a first intermediate code associated with the first programming language, wherein the first intermediate code being interpretable by at least one first intermediate code virtual machine;

converting the first intermediate code into a second intermediate code, wherein the step of converting comprises:

[at least one of the steps of:

a)] recording all jumps and their destinations in the original byte codes;

performing a conversion operation selected from the group:

[b)] converting specific byte codes into equivalent generic byte codes[or vice-versa];

[c)] modifying byte code operands from references using identifying strings to references using unique identifiers; and

[d)] renumbering byte codes in [a compiled format] the compiled form to equivalent byte codes in an instruction set supported by an interpreter on the integrated circuit card[a format suitable for interpretation]; and

relinking jumps for which the destination address is [effected] affected by the conversion operation[step a), b), c), or d)];

wherein the second intermediate code is interpretable within the set of resource constraints by at least one second intermediate code virtual machine; and

loading the second intermediate code into the memory of the microcontroller.

79. The method of programming a microcontroller of Claim 78 wherein the step of converting further comprises:

associating an identifying string for objects, classes, fields, or methods; and

mapping such strings to unique identifiers.

80. The method of Claim 79 wherein the step of mapping comprises the step of mapping strings to integers.

81. The method of Claim 80 wherein the step of loading the second intermediate code into the memory of the microcontroller further comprises checking the second intermediate code prior to loading the second intermediate code to verify that the second intermediate code meets a predefined integrity check and that loading is performed according to a security protocol.

82. The method of Claim 81 wherein the security protocol requires that a particular identity must be validated to permit loading prior to the loading of the second intermediate code.

83. The method of Claim 81 further characterized by providing a decryption key and wherein the security protocol requires that the second intermediate code is encrypted using a loading key corresponding to the decryption key.

84. (Amended) An integrated circuit card for use with a terminal, comprising:

a communicator configured to communicate with the terminal;

a memory storing:

an application derived from a program written in a high level programming language format wherein the application is derived from a program written in a high level programming language format by first compiling the program into a compiled form and then converting the compiled form into a converted form, the converting step including the steps of:

recording all jumps and their destinations in the original byte codes;

modifying byte code operands from references using identifying strings to references using unique identifiers;

[recording all jumps and their destinations in the original byte codes;]

converting specific byte codes into equivalent generic byte codes[or vice-versa];[and]

renumbering byte codes in [a compiled format] the compiled form to equivalent byte codes in an instruction set supported by an interpreter on the integrated circuit card[a format suitable for interpretation]; and

adjusting jump destinations that are affected by at least one of the steps of modifying byte code operands, converting specific byte codes into equivalent generic byte codes, or renumbering byte codes; and

an interpreter operable to interpret such an application derived from a program written in a high level programming language format; and

a processor coupled to the memory, the processor configured to use the interpreter to interpret the application for execution and to use the communicator to communicate with the terminal.

85. (Amended) A method for use with an integrated circuit card and a terminal, comprising:

storing an interpreter operable to interpret programs derived from programs written in a high level programming language and an application derived from a program written in a high level programming language format in a memory of the integrated circuit card wherein the application is derived from a program written in a high level programming language format by first compiling the program into a compiled form and then converting the compiled form into a converted form, the converting step including:

recording all jumps and their destinations in the original byte codes;

modifying byte code operands from references using identifying strings to references using unique identifiers;

[recording all jumps and their destinations in the original byte codes;]

converting specific byte codes into equivalent generic byte codes [or vice-versa]; [and]

renumbering byte codes in [a compiled format] the compiled form to equivalent byte codes in an instruction set supported by an interpreter on the integrated circuit card[a format suitable for interpretation]; and

adjusting jump destinations that are affected by at least one of the steps of modifying byte code operands, converting specific byte codes into equivalent generic byte codes, or renumbering byte codes; and

using a processor of the integrated circuit card to use the interpreter to interpret the application for execution; and

using a communicator of the card when communicating between the processor and the terminal.

86. (Amended) An integrated circuit card for use with a terminal, comprising:

a communicator configured to communicate with the terminal;

a memory storing:

applications, each application derived from applications having a high level programming language format, and

an interpreter operable to interpret applications derived from applications having a high level programming language format wherein the application is derived from a program written in a high level programming language format by first compiling the program into a compiled form and then converting the compiled form into a converted form, the converting step including the steps of:

recording all jumps and their destinations in the original byte codes;

modifying byte code operands from references using identifying strings to references using unique identifiers;

[recording all jumps and their destinations in the original byte codes;]

converting specific byte codes into equivalent generic byte codes[or vice-versa]; [and]

renumbering byte codes in [a compiled format] the compiled form to equivalent byte codes in an instruction set supported by an interpreter on the integrated circuit card[a format suitable for interpretation]; and

adjusting jump destinations that are affected by at least one of the steps of modifying byte code operands, converting specific byte codes into equivalent generic byte codes or vice versa, and renumbering byte codes; and

a processor coupled to the memory, the processor configured to:

a.) use the interpreter to interpret the applications for execution,

b.) use the interpreter to create a firewall to isolate the applications from each other, and

c.) use the communicator to communicate with the terminal.

87. (Amended) A microcontroller comprising:

a memory storing:

a derivative application derived from an application having a class file format wherein the application is derived from an application having a class file format by first compiling the application having a class file format into a compiled form and then converting the compiled form into a converted form, the converting step including:

recording all jumps and their destinations in the original byte codes;

converting specific byte codes into equivalent generic byte codes[or vice-versa]; [and]

renumbering byte codes in [a compiled format] the compiled form to equivalent byte codes in an instruction set supported by an interpreter on the integrated circuit card; [a format suitable for interpretation,] and

adjusting jump destinations that are affected by at least one of the steps of converting specific byte codes into equivalent generic byte codes and renumbering byte codes; and

an interpreter configured to interpret applications derived from applications having a class file format; and

a processor coupled to the memory, the processor configured to use the interpreter to interpret the derivative application for execution.

88. (New) A method of programming a microcontroller having a memory and a processor operating according to a set of resource constraints, the method comprising the steps of:

inputting an application program in a first programming language;

compiling the application program in the first programming language into a first intermediate code associated with the first programming language, wherein the first intermediate code being interpretable by at least one first intermediate code virtual machine;

converting the first intermediate code into a second intermediate code, wherein the step of converting comprises:

recording all jumps and their destinations in the original byte codes;

modifying byte code operands from references using identifying strings to references using unique identifiers; and

renumbering byte codes in the compiled form to equivalent byte codes in an instruction set supported by an interpreter on the integrated circuit; and

relinking jumps for which the destination address is affected by the conversion operation;

wherein the second intermediate code is interpretable within the set of resource constraints by at least one second intermediate code virtual machine; and

loading the second intermediate code into the memory of the microcontroller.

89. (New) The method of programming a microcontroller of Claim 88 wherein the step of converting further comprises:

associating an identifying string for objects, classes, fields, or methods; and

mapping such strings to unique identifiers.

90. (New) The method of Claim 89wherein the step of mapping comprises the step of mapping strings to integers.

91. (New) A method of programming a microcontroller having a memory and a processor operating according to a set of resource constraints, the method comprising the steps of:

inputting an application program in a first programming language;

compiling the application program in the first programming language into a first intermediate code associated with the first programming language, wherein the first intermediate code being interpretable by at least one first intermediate code virtual machine;

converting the first intermediate code into a second intermediate code, wherein the step of converting comprises:

recording all jumps and their destinations in the original byte codes;

performing a conversion operation comprising:

modifying specific byte codes into equivalent generic byte codes; and

relinking jumps for which the destination address is affected by the conversion operation;

wherein the second intermediate code is interpretable within the set of resource constraints by at least one second intermediate code virtual machine; and

loading the second intermediate code into the memory of the microcontroller.

92. (New) A method of programming a microcontroller having a memory and a processor operating according to a set of resource constraints, the method comprising the steps of:

inputting an application program in a first programming language;

compiling the application program in the first programming language into a first intermediate code associated with the first programming language, wherein the first intermediate code being interpretable by at least one first intermediate code virtual machine;

converting the first intermediate code into a second intermediate code, wherein the step of converting comprises:

recording all jumps and their destinations in the original byte codes;

performing a conversion operation comprising:

renumbering byte codes in the compiled form to equivalent byte codes in an instruction set supported by an interpreter on the integrated circuit card; and

relinking jumps for which the destination address is affected by the conversion operation;

wherein the second intermediate code is interpretable within the set of resource constraints by at least one second intermediate code virtual machine; and

loading the second intermediate code into the memory of the microcontroller.

93. (New) An integrated circuit card for use with a terminal, comprising:

a communicator configured to communicate with the terminal;

a memory storing:

an application derived from a program written in a high level programming language format wherein the application is derived

from a program written in a high level programming language format by first compiling the program into a compiled form and then converting the compiled form into a converted form, the converting step comprising:

recording all jumps and their destinations in the original byte codes;

performing a conversion operation including modifying byte code operands from references using identifying strings to references using unique identifiers; and

relinking jumps for which the destination address is affected by the conversion operation; and

an interpreter operable to interpret such an application derived from a program written in a high level programming language format; and

a processor coupled to the memory, the processor configured to use the interpreter to interpret the application for execution and to use the communicator to communicate with the terminal.

94. (New) An integrated circuit card for use with a terminal, comprising:

a communicator configured to communicate with the terminal;

a memory storing:

an application derived from a program written in a high level programming language format wherein the application is derived from a program written in a high level programming language format by first compiling the program into a compiled form and then converting the compiled form into a converted form, the converting step comprising:

recording all jumps and their destinations in the original byte codes;

performing a conversion operation including:

converting specific byte codes into equivalent generic byte codes; and

renumbering byte codes in the compiled form to equivalent byte codes in an instruction set supported by an interpreter on the integrated circuit card; and

relinking jumps for which the destination address is affected by the conversion operation; and

an interpreter operable to interpret such an application derived from a program written in a high level programming language format; and

a processor coupled to the memory, the processor configured to use the interpreter to interpret the application for execution and to use the communicator to communicate with the terminal.

REMARKS:

Status of the Claims:

In the Office Action mailed December 21, 2007 the Examiner rejected claims 1-87. Patentee amends Claims 1, 31, 58, 64, 65, 78, 84, 85, 86, and 87 and adds new Claims 88-94. Claims 1-94 are now pending in the application.

Explanation of Support for Claim Amendments and Explanation of Narrowing Nature of the Amendments

Patentee amends Claims 1, 31, 58, 64, 65, 78, 84, 85, 86, and 87. The amendments to Claims 1, 31, 58, 64, and 65 follow a similar pattern. Consider Claim 1 as an example:

1. An integrated circuit card for use with a terminal, comprising:

a communicator configured to communicate with the terminal;

a memory storing:

an application derived from a program written in a high level programming language format wherein the application is derived from a program written in a high level programming language format by first compiling the program into a compiled form and then converting the compiled form into a converted form, the converting step [including at least one step selected from a group consisting of]comprising:

recording all jumps and their destinations in the original byte codes;

performing a conversion operation selected from the group:

converting specific byte codes into equivalent generic byte codes[or vice-versa];

modifying byte code operands from references using identifying strings to references using unique identifiers; and

renumbering byte codes in [a compiled format] the compiled form to equivalent byte codes in an instruction set supported by an interpreter on the integrated circuit card[a format suitable for interpretation]; and

relinking jumps for which the destination address is affected by the conversion operation; and

an interpreter operable to interpret such an application derived from a program written in a high level programming language format; and

a processor coupled to the memory, the processor configured to use the interpreter to interpret the application for execution and to use the communicator to communicate with the terminal.

The combination of the deletion "[including at least one step selected from a group consisting of]" and addition "performing a conversion operation selected from the group:" causes the element "recording all jumps and their destinations in the original byte codes" to be moved outside of the Markush group introducing the conversion process. Thus, this addition narrows the claim and is supported in the original claim.

Removal of the phrase "[or vice versa]" only narrows the claim.

The amendments to the element "renumbering byte codes in [a compiled format] the compiled form to equivalent byte codes in an instruction set supported by an interpreter on the integrated circuit card[a format

suitable for interpretation];" are supported, at least, by Col. 11, Lines 4-23. The edits narrow the limitation.

The added element "relinking jumps for which the destination address is affected by the conversion operation" is supported by Claims 65 and 78 in the '317. Furthermore, this limitation is supported, at least, by Col. 11, lines 37-44. The addition of this limitation narrows the claims.

Therefore, for the foregoing reasons, the amendments to Claim 1 are supported by the issued patent and narrows the claim. Accordingly, the amendments satisfy the requirements for an amendment set forth in 37 CFR 1.530.

Claims 31, 58, 64, and 65 are amended in a similar fashion to Claim 1 and satisfy the requirements of 37 CFR 1.530 for the same reasons.

Claim 78 is amended in a similar fashion. However, the last element "relinking of jumps" was already present in Claim 78. Accordingly, the notes above related to Claim 1 also apply to Claim 78.

Claim 84 is amended by reordering two steps, recording jumps modifying byte code operands; removing "vice versa" in the step of converting specific byte codes to generic equivalents and rewording the step of renumbering byte codes as described above in the discussion of Claim 1. Finally, the adjusting jump destinations limitation has been added to the claim. These limitations are all found in similar edits in Claim 1 and the relevant discussion showing support and narrowing of claim scope vis-à-vis Claim 1 is incorporated here by reference.

Claims 85, 86, and 87 provide similar amendments to those in Claim 84 and satisfy the requirements for claim amendments for the same reasons as those given in support of the amendments to Claim 84.

Accordingly, all the amendments comply with the requirements for making amendments in a reexamination procedure. Patentee respectfully request entry of the amendments.

New Claims

Patentee submits new claims 88 through 94. These claims are supported by Claims 1 and 78 in the original and for the reasons given in support of the amendments to Claims 1 and 78. They provide claims in which the Markush group found in the issued version of Claims 1 and 78 have been replaced with elements found in those Markush groups, respectively. Accordingly, these claims are inherently narrower than the claims upon which they are based and are supported by the language found in those issued claims, respectively.

In Regard to General Introductory Remarks in Office Action

In the Office Action at Section IV, the Examiner summarizes the invention as "the '805 (sic, should be '317) patent is drawn to a method and apparatus that includes an integrated circuit card with a memory that stores an interpreter and an application that has a high level programming language format for use with a terminal (e.g., a 'smart card')." (Office Action, page 5, lines 2-4) That summary continues with a description of the creation of programs for the smart cards using high-level languages such as Java using mainstream programming tools. Up to that point, the Examiner's summary appears correct.

However, the Examiner's summary concludes with several incorrect statements:

(1) "The high level programming language format of the application can a class file format and may have a Java programming format." (Office Action, Page 5, lines 12-13).

(2) "That is, the invention as claimed simply combines the well-known Java programming language, with well-known hardware (e.g., a microcontroller "smart card"). (Office Action, Page 5, lines 13-15)

(3) "Inherent to the Java VM is that a converter accepts the compiled form in standard Java class file format and produces output in a form for interpretation by the native machine." (Office Action, Page 5, lines 21-22)

Let's consider each of these statements in turn:

(1) A more correct statement would be that a high-level language application is compiled and, in the case of a Java application, into a class file format, i.e., the Java class file format. The distinction may seem subtle. However, it is the compiled output that is used by the converter that, as will be seen hereinbelow, produces a reduced intermediate file, e.g., in the case of Java, a card class file in a format suitable for interpretation on the smart card.

(2) Our main objections to this statement are the implied limitation to the Java programming language and any inferences that a reader may draw from the word "simply." As will be discussed in greater detail below,

there is nothing *simple* about taking a program created by a standard high-level language development environment, e.g., a Java development environment, and making it possible for that program to execute on a smart card.

(3) A Java VM does not convert Java class file format into a form for interpretation by a native machine. Rather, a Java VM is a program that would have been compiled to execute on particular machines. However, there is nothing implied in doing so that the JVM provides for any form of transformation of the class file format into a form in which that program would execute on the "native machine." The Java virtual machine is an interpreter that interprets the Java class file against the architecture specified for the JVM. No conversion from the compiled format to a format for interpretation by the native machine takes place in the interpretation of the class files.

Summary of the Invention

Patentee provides the following summary of the technology presented in '317 to assist the Examiner in evaluating the claims with respect to the prior art.

Prior to the development of the technology set forth in the '317 patent, program development for smart cards suffered because of the limitations of programming tools available for smart card program development. Programming for smart cards relied on vendor and card-specific tools and

languages. Programming of new applications generally had to be performed with the assistance of smart card vendors. Therefore, it was difficult to provide applications that ran on different cards from different vendors. Program development suffered due to limitations in resources available and limitations in the imagination of smart card vendors. While smart cards provided many benefits, for example, by being an extraordinarily secure platform, the smart card business was held back by the difficulty in developing new applications.

This situation is summarized in one of the references cited in the Office Action: Patrice Peyret, *Application-Enabling Card Systems with Plug-and-Play Applets*, in Proceedings of the 1996 Smart Card Convention, Quality marketing services Ltd, Peterborough, UK (Feb. 1996) (Reference 13) (hereinafter "*Peyret Article*" (to clearly differentiate from *Peyret '884*)). Peyret states that "[m]ost Smart Card Systems today suffer from long development times, reduced interoperability, and poor upgradeability, as the tools for creating card and terminal software remain too crude and fragmented for most programmers." (*Peyret Article*, Page 51, lines 3-5). In Zhiquen Chen, *Java Card Technology for Smart Cards*, published by Sun Microsystems, these challenges are further described:

> "Developing a smart card application traditionally has been a lengthy and difficult process. ... Most smart card development tools are built by the smart card manufacturers using generic assembly language tools and dedicated hardware emulators obtained from silicon chip vendors. It has been virtually impossible for third parties to develop applications independently and sell them to issuers. Therefore, developing smart card applications has been limited to a

group of highly skilled and specialized programmers who have intimate knowledge of the specific smart card hardware and software.

Because there are no standardized high-level application interfaces available in smart cards, application developers need to deal with very low-level communication protocols, … Upgrading software or moving applications to a different platform is particularly difficult or impossible."

Zhiquen Chen, *Java Card Technology for Smart Cards, Architecture and Programmer's Guide*, Sun Microsystems, page 7 (2000) (Attached hereto as Appendix A).

To solve the development problem, the inventors set out to devise a mechanism by which program development could be performed using standard high-level programming languages. One benefit would be the high quality of such tools and languages. Another benefit, especially in the case of Java, is the availability of security features built into the programming language. Yet another benefit would be the possibility to test the programs by running the programs on the computers on which the programs were developed before loading the programs onto the smart card. Programs would also be more easily portable from one card to another. These advantages are just examples, there may be others.

However, using Java as an example, providing Java technology onto smart cards would be very challenging due to the size constraints of smart cards as contrasted to the minimum requirements of Java. The Java run time environment, the JRE, requires a system that has a minimum of 32MB bytes of memory, 125 MB of free disk space. However, in 1996 smart cards only had 512 bytes, not kilobytes or megabytes, just 512 bytes of RAM and 4K

bytes of EEPROM. The ratio between 32 MB and 512 bytes is 65536. Visually that difference is depicted in Figure 1 below.

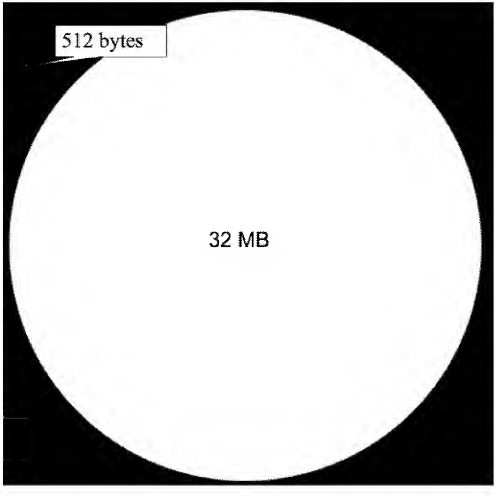


Figure 1.

Similarly, the ratio between 125MB and 4K is 32,000, which would visually be very similar to the size difference for the RAM shown in Figure 1.

Smart cards are often used for applications such as authentication, cryptography, and managing mobile phone subscriber profiles. These

applications, naturally, are quite large. These applications typically compile into files much larger than the pragmatically available space in a smart card. Thus, the problem of fitting them onto a smart card would be rather challenging.

Also, the intermediate language interpreter, e.g., the JVM, would also have to fit on the smart card.

To address the program development deficiencies for smart card programming, the inventors devised a system in which "[s]mart card applications may be created in high level languages such as Java and Eiffel, using powerful mainstream program development tools. New applications can be quickly prototyped and downloaded to a smart card in a matter of hours without resorting to softmasks." ('317, Col. 3, line 67-Col. 4, line 2).

Because the invention, which indeed is novel, non-obvious, and highly useful, provides, at least, one additional tool or step, it is possible to use this existing technology (off-the-shelf technology at that) to develop smart card applications; thereby achieving the goal of providing the use of existing high-level languages and development tools to develop applications for devices for which applications previously had to be developed using "crude" tools (to use Peyret's terminology).

To deal with the resource constraints of smart cards, the solution included creating a reduced class file specification (the card class file), a reduced interpreter that would interpret programs in the reduced class file specification, and a special program to convert between the compiled output of the high-level language development environment and the reduced class file specification, the *converter*, the at least one additional tool or step added by the inventors.

As claimed, the converter performs certain operations. Applicants have amended the independent claims to more clearly recite the subject

matter of the invention. Consider amended claim 78 as an example. Claim 78 recites:

"78. A method of programming a microcontroller…

… converting comprises:

recording all jumps and destinations in the original byte codes;

performing a conversion operation selected from the group:

converting specific byte codes into equivalent generic byte codes;

modifying byte code operands from references using identifying strings to references using unique identifiers;

renumbering byte codes in the compiled form to equivalent byte codes in an instruction set supported by an interpreter on the integrated circuit card;

relinking jumps for which the destination address is affected by the conversion operation

…"

Thus, the converter performs, at least, recording jumps, a conversion operation, and relinking jumps. The conversion operator is selected from the group having the members converting specific byte codes into equivalent generic byte codes, modifying byte code operands from references using identifying strings to references using unique identifiers, and renumbering byte codes in the compiled format into byte codes suitable for interpretation. These steps are likely to cause a change in the converter form with respect to

the compiled form. Thus, the transformation steps may require adjustment to jumps that are affected by the transformation steps. Therefore, any jumps and their destinations are recorded and subsequent to the conversion step, the jumps are adjusted.

These operations reduce the size of the program and/or make it possible for the interpreter to be much smaller, for example, by virtue of supporting a reduced instruction set.

35 USC 103(a)

In the Office Action, the Examiner rejected Claims 1, 3-6, 8-22, 24-28, 31-35, 54-60, 63-65, 67, 71-78, 80, 81, and 84-87 under 35 USC 103(a) as being unpatentable over U.S. Patent 5,923,884 to *Peyret et al.* (*Peyret '884*) in view of U.S. Patent 5,668,999 to Gosling (*Gosling '999*)[1]. Patentee traverses the rejection as it may be applied to the amended claims and the new claims presented herein.

As noted above, Patentee has amended the independent claims herein to more clearly recite the subject matter of the invention. In the Patent version of the claims, the independent claims (using Claim 78 as a model) set out the limitations for a conversion step as the combination *recording jumps and destinations, a conversion operation, and relinking jumps.* In the patented version of the independent claims the step of *recording jumps* was made part of a *Markush* group from which the *conversion operation* was

[1] Patentee notes that the office action also refers to U.S. Patent 5,367,685 to Gosling (Office Action, page 15 (without being enumerated in the table of references set out in office action, Office Action, pages 3-4)) and J.Gosling, "Java Intermediate Bytecodes", ACM SIGPLAN Workshop, IR '95, 1995 ACM (Office Action, Pages 4 and 11). Patentee has inferred from the pinpoint cite pages 7:33-39 and 2:30-32 that Gosling '999 was intended as Gosling '685 does not have a column 7.

selected. *Gosling* was cited against the *recording jumps* limitation and in the Office Action it was rejected because *Gosling* allegedly taught that one element. Patentee agrees with that legal principle. However, as amended, the *Markush* group no longer contains the *recording jumps* limitation. Therefore, the importance of *Gosling* in the rejection has been reduced as merely relevant to one of several limitations. Patentee discusses *Gosling* hereinbelow when discussing the patentably distinct features of the specific conversion operations. However, Patentee wishes to first address the teaching of *Peyret '884* and its failing to teach or suggest the claimed invention, singly, or in combination with the other cited reference.

The Examiner sets out the rejection of the independent claims as unpatentable over *Peyret '884* in view of *Gosling* on pages 7 through 15 of the office action. Patentee's understanding of the rejection is that the following logic is applied:

- Peyret teaches the use of high-level languages that are compiled.
- The compiled output files are linked.
- Patentee claims conversion.
- Conversion is the same as linking.
- Since Peyret teaches compiling high-level languages, and compiled output files are linked, it follows that "a functionally equivalent conversion process would be inherent in the Peyret reference." (Office Action, Page 7, lines 21-22).
- And, further that each of the specific limitations of the conversion step are taught by various references that may be combined with Peyret.

Patentee disagrees with this line of reasoning.

In the Office Action, the Examiner stated that "it is inherent [in *Peyret '884*] that applets are written in a high level programming language and then compiled into a compiled form before execution." (Office Action, Page 7, lines 13-14). Patentee reserves the right to argue on appeal that the premise that Peyret teaches the use of high-level language is incorrect. However, Patentee does not believe that it is necessary to address that issue to illustrate that ultimate conclusion that "a functionally equivalent conversion process would be inherent in the Peyret reference" is incorrect.

Turning now to the notion that conversion and linking are the same. The Examiner incorrectly states that "the '317 specification appears to reveal that the conversion process simply amounts to the well known process of 'linking', where one or more objects are generated [by] a compiler and assembled into a single executable file." (Office Action, Page 7, lines 17-19). That is not correct.

First of all, the outputs linkers and converters are different. Linkers produce load modules ("The linker's function is to collect procedures translated separately and link them together to be run as a unit, usually called an absolute load module." (*Tanenbaum*, Reference 11, Page 418, lines 3-4)). Patentee requests that the Examiner take judicial notice of that an absolute load module is a program wherein memory references have been resolved from symbolic addresses into absolute addresses and in which intra-module references have been resolved.

The output from the converter, on the other hand, in the case of Java is a class file in which information distributed over several Java class files have been coalesced into one file ('317, Col. 9, lines 19-23). "Coalesced" does not mean resolving symbolic references. Nowhere in the claims or in the specification of '317 is there a suggestion that the card class file is an absolute load module. None of the claimed converter steps deal with

resolving addresses based on relative memory locations of the respective object modules or intra-module references. Thus, it would be incorrect to infer that the converter produces an absolute load module.

Second, the steps performed by the loader and converter are different. That should not be surprising considering that the outputs from the converter and a linker are different.

Patentee will describe the conversion operations in greater detail hereinbelow. However, to contrast with linking, consider that the conversion step is claimed as "recording all jumps and destinations in the original byte codes; performing a conversion operation selected from the group: converting specific byte codes into equivalent generic byte codes; modifying byte code operands from references using identifying strings to references using unique identifiers; renumbering byte codes in the compiled form to equivalent byte codes in an instruction set supported by an interpreter on the integrated circuit card; and relinking jumps for which the destination address is affected by the conversion operation." For the sake of brevity of discussion, let's refer to these steps as "recording jumps and destinations", "performing a conversion operation", "relinking jumps", wherein the "conversion operation" is selected from "modifying specific byte codes for generic equivalents", "renumbering byte codes" and "replacing operand strings with IDs". None of these steps are performed by a linker. This difference may be even more readily apparent when considering the discussion below in which the specific conversion operations recited in the claims are contrasted against the prior art cited against the specific conversion operation sub-steps.

For descriptions of linking, Patentee turns to the references provided by the Examiner: *Presser and White*[2](Reference 5), *Tanenbaum* (Reference 11)[3], *Srivastava*[4](Reference 1), and *Unix Programmer's Manual*[5](Reference 9). Each of these references describes the classical notion of linking object files that have symbolic references. As noted by the Examiner, "*Presser and White*, for example, describe a linker whose responsibility it is 'to combine the input subprograms into a single re-locatable output module in which all external references have been resolved." (Office Action, Page 8, lines 12-14). Thus, a linker, in that sense, is a program for combining subprograms and resolving external references. *Tanenbaum* states that a "linker merges the separate address spaces of the object modules" (Page 421, lines 1-2), "assigns a load address to each object module" (Page 421, line 4), "adds a relocation constant [to each memory address reference" (Page 421, lines 5-7), and resolves external references (Page 421, lines 8-9). None of these steps are performed by the converter. None of these steps are claimed as converter operations. Conversely, linkers do not perform any of the claimed steps of "recording jumps and destinations", "performing a conversion operation", "relinking jumps", wherein the "conversion operation" is selected from "modifying specific byte codes for generic equivalents", "renumbering byte codes" and "replacing operand strings with IDs". That is not surprising since none of these steps are useful to produce an absolute load module.

Returning now to *Peyret '884*. The Examiner stated that "a functionally equivalent conversion process would be inherent in the *Peyret*

---

[2] Leon Presser and John R. White, Linkers and Loaders, COMPUTING SURVEYS, VOL. 4, No. 3 (September 1972)
[3] Andrew S. Tanenbaum, Structured Computer Organization (3d ed. 1990)
[4] U.S. Patent 5,457,799
[5] B.W. Kernighan and M.D. McIlroy, UNIX Programmer's Manual, Bell Telephone laboratories, Inc., Murray Hill (7th Ed. 1979)

reference" (Office Action, Page 7, lines 21-22). Patentee reminds the Examiner that "[to] establish inherency, the extrinsic evidence 'must make clear that the missing descriptive matter is necessarily present in the thing described in the reference, and that it would be so recognized by persons of ordinary skill. Inherency, however, may not be established by probabilities or possibilities. The mere fact that a certain thing may result from a given set of circumstances is not sufficient.'" *In re Robertson*, 169 F.3d 743, 745, 49 USPQ2d 1949, 1950-51 (Fed. Cir. 1999), *cited in* MPEP 2112 IV.

That *Peyret '884* teaches a system in which a conversion process is not inherent may be understood from what Peyret disclosed in the *Peyret article*[6] (Reference 13)which preceded the filing date of '884 by approximately six months. The *Peyret article* states that "creating a general-purpose language which would make it easy for application providers to implement efficiently whichever cryptographic routines they liked and load them at will inside cards and terminals would be asking for trouble." (*Peyret article,* Reference 13, Page 62, lines 43-46). Java is precisely a general-purpose language that makes it easy to program any functionality including cryptographic routines. Thus, Peyret advised against providing a system such as the one suggested and claimed by Patentee.

Furthermore, the *Peyret article* directly taught against doing what the inventors of '317 sought to do, succeeded in doing and have claimed as their invention. In the *Peyret article* it is stated that "[t]he wrong approach to interpretive structures in card systems would be to pick an existing interpreted computer language, and hammer it into cards and terminals."

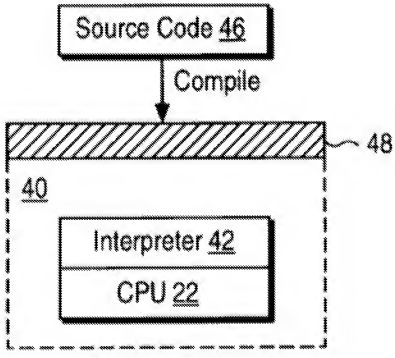
[6] Ref. 13: Patrice Peyret, Application-Enabling Card Systems with Plug-and-Play Applets, in Proceedings of the 1996 Smart Card Convention, Quality marketing services Ltd. Peterborough UK, (Feb 1996)

(*Peyret article*, Reference 13, Page 59, lines 11-12). Patentee posits that that is a direct teaching against the claimed invention.

Patentee wishes to make it clear that patentee does not intend to mislead by referring to the *Peyret article* rather than *Peyret '884*. However, as inherency requires that the missing descriptive matter is necessarily present, any reference that establishes that there are alternatives would be relevant to countering the position that something is inherent. That Peyret had something very different in mind as an alternative is further illustrated by his description of a specialized language for writing smart card applications, namely, that "[t]he interpreted language for Applets is a transaction-oriented language, very much optimized and dedicated to the implementation of trusted exchanges between cards and terminals." - (*Peyret article*, Reference 13, Page 66, lines 33-36). In other words, Peyret suggested the creation and use of a special purpose language for smart card applications. That language would be an alternative to using a general-purpose high-level language. Patentee requests that the Examiner take judicial notice that it would be non-intuitive and counter-productive to design such a language so that a conversion operation would be required. There would be no motivation to design the special purpose language such that programmers would be required to execute the conversion process. It would be much more straightforward to use the classic approach of simply compiling the special purpose language into an object file for linking and loading into the memory. In other words, a conversion process would not be inherent in *Peyret '884*.

If *Peyret '884* did not include a conversion process, how does *Peyret '884* create and load programs onto smart cards? Figure 2 of *Peyret '884* illustrates this operation:

FIG. 2



Peyret '884 Figure 2.

In the accompanying text to Figure 2, *Peyret '884* describes the operation as "[t]o execute an applet on an interpreter, as shown, source code 46 of an applet is compiled into a byte code 48" (*Peyret '884*, Reference 4, Col. 5, Lines 59-62)." There is no implication that a conversion process is being performed.

The very issue of whether *Peyret '884* suggests a conversion process was addressed by the Board of Appeals in the continuation application U.S. Patent 7,117,485:

> "There is no disclosure in Peyret ['884] which suggests that a conversion as claimed occurs between the byte code 48 and the interpreter 42. The command dispatcher or reduced interpreter suggested by Peyret is described as a substitute for the interpreter and virtual machine. There is no suggestion, however, that the reduced interpreter operates to compile an application having a class file format and then convert the compiled form into a converted form." (*Ex Parte Wilkinson*, Application 10/037,390, Decision on Appeal, Page 7.")

The claims in *'485* are broader than the claims in '317 as the conversion process there is set forth without specifically reciting substeps involved in performing a conversion. Thus, the issue faced in '485 is directly on point here.

To conclude, the *Peyret article* clearly indicates that an alternative to general-purpose high-level interpreted languages may be used for programming smart cards, namely the special purpose programming language contemplated therein. Therefore, it would be probable, or at least a possible alternative, that the system in *Peyret '884* used such a language. Because there would be no motivation to create that language such that conversion would be necessary, it follows that it is *not* inherent and not even likely that *Peyret '884* required a conversion process. Furthermore, that is the position taken by the Board of Appeals in *Ex Parte Wilkinson*, Application 10/037,390, now U.S. Patent 7,117,485.

Conversion Operation

Turning now to the specific operations of the conversion operation and the references cited against each of those elements. Claim 78 recites

"…recording all jumps and their destinations in the original byte codes; performing a conversion operation selected from the group:

converting specific byte codes into equivalent generic byte codes;

modifying byte code operands from references using identifying strings to references using unique identifiers; and

renumbering byte codes in the compiled form to equivalent byte codes in an instruction set supported by an interpreter on the integrated circuit card; and

relinking jumps for which the destination address is affected by the conversion."

The other independent claims recite analogous limitations.

The *recording of jumps/relinking of jumps* steps are more readily understood and explained after the conversion operation, which, in the claims, is located between these two operations. Therefore, the conversion operation is discussed first.

The conversion operation is set forth in the claims as being selected from a group including "converting specific byte codes into equivalent generic byte codes", "modifying byte code operands from references using identifying strings to references using unique identifiers" and "renumbering byte codes in the compiled form to equivalent byte codes in an instruction set supported by an interpreter on the integrated circuit card." These operations are not taught or suggested by the cited prior art.

Converting specific byte codes into equivalent generic byte codes.

The operation of "converting specific byte codes into equivalent generic byte codes" is illustrated, for example, in Figure 7 of '317:

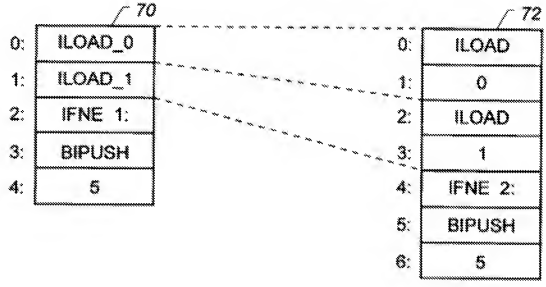


FIGURE 7

Figure 7 from '317

The specification for Java byte codes provide for many specific instructions in which an operand is implied. In Figure 7 of '317 two such instructions are shown, namely, ILOAD_0 and ILOAD_1. The ILOAD_x instructions load the x^{th} argument, respectively. Thus, ILOAD_0 loads the zeroth argument and ILOAD_1, the first argument. The operand is implied.

The generic alternative is the instruction ILOAD. The ILOAD instruction takes one operand, namely the index of the argument to be loaded. Thus, to load the zeroth argument, the program could include the instruction ILOAD 0.

In the step of "converting specific byte codes into equivalent generic byte codes" specific byte codes, such as ILOAD_0 and ILOAD_1, are replaced with the equivalent generic instruction, i.e., in the case of the example of Figure 7, ILOAD 0 and ILOAD 1, respectively.

The Examiner states in the Office Action that "*Tanenbaum*[7] at 28-30, for example, teaches numerous different specific examples of situations where one or a set of instructions (byte codes) is replaced by different but equivalent instructions." (Office Action, Page 12, Lines 20-21). Applicants disagree that *Tanenbaum* teaches or suggests "converting specific byte codes into equivalent generic byte codes."

Pages 28-30 of *Tanenbaum* is a replacement table used for peephole optimization ("... performing peephole optimizations on the intermediate code. A peephole optimization is one that *replaces a sequence* of consecutive instructions by a semantically equivalent but *more efficient sequence*. The sequences to be matched and their replacements are described in a driving table." (*Tanenbaum*, Reference 10, Page 22, Lines 16-19). The driving table, which is found in the cited passage on Pages 28-30 of *Tanenbaum* provides sequences of code that may be replaced by other sequences of code that are more efficient. Consider, for example, the first entry in the table:

1. (225) LOC A LOC B ADD => LOC A+B

Tanenbaum deals with a stack oriented architecture in which the ADD instruction adds the two top entries on the stack. Thus, LOC is the load constant instruction, which loads its operand onto the stack. Thus, the sequence LOC A, LOC B, ADD causes the addition of the constants A and B. Naturally, that sequence can be replaced for the more efficient LOC A+B. However, that is not a replacement of a specific instruction for a generic

[7] ANDREW S. TANENBAUM, HANS VAN STAVERN and JOHAN W. STEVENSON, Using Peephole Optimization on Intermediate Code, ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS, VOL. 4, No. 1 (January 1982)

equivalent. Rather, on both sides of the replacement the same instruction appears.

In the entire table from *Tanenbaum* pages 28-30 there is not one instance of a replacement of a specific instruction with a generic equivalent. Which is not surprising because the substitution of specific instructions with generic equivalent instructions actually slow down programs.

It should be further noted that *Tanenbaum* and Patentee are not attempting to solve the same issue. *Tanenbaum* is explicitly after code optimization. Performing the claimed substitution actually slows down the code because of additional operations required by the interpreter. Conversely, Patentee performs the substitution, at least, for the purpose of reducing the instruction set supported by the virtual machine so that a smaller virtual machine may be utilized on the smart card. *Tanenbaum's* suggested replacements do not allow for removing instructions from the virtual machine. Thus, there would be no motivation for a person trying to devise a reduced interpreter based on supporting a reduced instruction set to implement *Tanenbaum's* suggested replacements.

For the foregoing reasons, *Tanenbaum* does not teach or suggest "converting specific byte codes into equivalent generic byte codes."

Renumbering byte codes

Claim 78 further recites "renumbering byte codes in the compiled form to equivalent byte codes in an instruction set supported by an interpreter on the integrated circuit card."

Figure 10 of '317 provides an illustrative example of this operation:

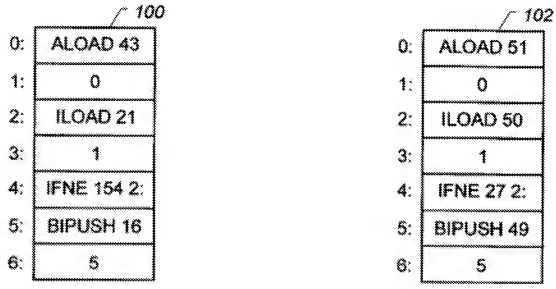


FIGURE 10

Figure 10 from '317

The code 100 is a pre-conversion Java class file. It contains instructions ALOAD, ILOAD, IFNE, and BIPUSH. The code 102 is the converted card class file corresponding to code 100. It also contains ALOAD, ILOAD, IFNE, and BIPUSH. However, in code 102 the byte codes used for these instructions are different from the ones in code 100. The reason for that is that in the card virtual machine a smaller instruction set is supported. It is advantageous to remove gaps in the instruction space and to group instructions so that similar instructions have byte codes that are close. Thus, as an example, the card virtual machine may associate the byte code 51 with ALOAD whereas the JVM associates 43 with ALOAD; and similarly ILOAD (50 rather than 21), IFNE (27 rather than 154), and BIPUSH (49 rather than 16).

To accommodate this difference in instruction spaces, the conversion process includes the step of "renumbering byte codes in the compiled form to equivalent byte codes in an instruction set supported by an interpreter on the integrated circuit card." Thus, in the example of Figure 10, ALOAD is renumbered from 43 to 51, and so on. However, in contrasting with the cited prior art it should be noted that the original byte code and the renumbered byte code refer to the same instruction. It is the *byte code* associated with the instruction that has been modified, not the instruction.

In the office action, the Examiner offers three references, namely, The Java Virtual Machine Specification Chapter 9 (Reference 14), *Tanenbaum* (Reference 10), and *Gosling '685* (Reference 2) with respect to the renumbering of byte codes limitation. Patentee disagrees that any of these references teach or suggest the *renumbering of byte codes*..

In regard to The Java Virtual Machine Specification, the Examiner stated that "[it] teaches the method of optimization by modifying compiled Java byte codes for better performance, including through the use of pseudo-instructions (so-called 'quick instructions') 'to do less work than the original instructions." (Office Action, Page 14, lines 5-7). Note that this statement implies an understanding that the quick-instruction and the original instruction do not perform the same task, otherwise it would not be possible for one to do less work than the other.

A difference between quick instruction and non-quick counterparts is that "[quick] instructions take advantage of loading and linking work done the first time the associated normal instruction is executed." (*Java Virtual Machine Specification*, Reference 14 Chapter 9.1, lines 16-17). "In all cases, the instructions with *_quick* variants reference the constant pool, a fairly costly operation. The *_quick* pseudo-instructions save time by exploiting the fact that, while the first time an instruction referencing the constant pool

must dynamically resolve the constant pool entry, subsequent invocations of that same instruction must reference the same object and need not resolve the entry again." (*ibid* at lines 23-26.). Thus, the two instructions are distinct instructions; one that accesses the constant pool, one that does not.

In contrast, the "renumbering byte codes in the compiled form to equivalent byte codes in an instruction set supported by an interpreter on the integrated circuit card" refer to the same instruction using different byte codes.

The Examiner's second reference, *Tanenbaum,* teaches code reordering. The Examiner points out that "*Tanenbaum* teaches a class of optimization steps that 'merely *reorders'* the instructions in each pattern without changing them.'" (Office Action, Page 15, lines 1-2). Code reordering can be understood from the table at the bottom of page 32 of *Tanenbaum*:

Original	1st opt. (119)	2nd opt. (2)	3rd opt. (97)	Final code (99)
LOV *A*	LOV *A*	LOV *A*	LOV *A*	INV *A*
LOC 6	LOC 6	LOC 1	INC	
ADD	LOC 5	ADD	STV *A*	
LOC 5	SUB	STV *A*		
SUB	ADD			
STV *A*	STV *A*			

The columns going from left to right represent successive optimizations from an original code sequence to a highly optimized code sequence. Consider first the replacements from Column 1 to Column 2. There the sequence of loading and performing arithmetic operations have been reshuffled into an equivalent sequence. No byte codes have been renumbered. In fact, no instructions have been changed; just the order of execution. There is no indication that *Tanenbaum's* instruction reordering changes the byte code by

which instructions are referred and can, therefore, not be deemed to teach or suggest "renumbering byte codes in the compiled form to equivalent byte codes in an instruction set supported by an interpreter on the integrated circuit card."

Furthermore, the Examiner stated that "the idea of rearranging instructions in order to reduce the space requirements, and replacing one Java byte code with another byte code better suited for execution on a different virtual machine, has long been taught by prior art in the field of computer science" (Office Action, Page 15, lines 3-5). The first part of that statement may be true, however, it is irrelevant. "[R]enumbering byte codes in the compiled form to equivalent byte codes in an instruction set supported by an interpreter on the integrated circuit card" (Claim 78) is not the same as "rearranging instructions".

Patentee disagrees with the assertion that "replacing one Java byte code with another byte code better suited for execution on a different virtual machine, has long been taught by prior art in the field of computer science" (Office Action, Page 15, lines 3-5). Such replacement of byte codes would be contrary to the fundamental portability goal of Java that Java programs should be executable on any virtual machine.

Apparently, to support the second notion of the assertion, the Examiner offered *Gosling '685* as an additional reference with respect to the *renumbering* element. Patentee disagrees that *Gosling '685* teaches or suggests anything relevant to "renumbering byte codes in the compiled form to equivalent byte codes in an instruction set supported by an interpreter on the integrated circuit card."

Gosling '685 deals with an operand replacement to avoid reinterpreting symbol operands that are determined to be constant index values into a data object (see *Gosling '685*, Figure 8). For example, a

symbolic reference "y" is rewritten as a numeric reference 2 to index the second slot in a data object. Performing that operation is not a renumbering of byte codes. Gosling's technique does not deal with instructions, it deals with operand replacement.

For the foregoing reasons, it is submitted that the cited prior art does not teach or suggest "renumbering byte codes in the compiled form to equivalent byte codes in an instruction set supported by an interpreter on the integrated circuit card."

Replacing Operand String Symbols with Identifiers

Claim 78 also recites "modifying byte code operands from references using identifying strings to references using unique identifiers" as one of the elements in the Markush group from which the conversion operation may be selected. Figure 3 of '317 is an illustration showing an example of the environment in which this operation takes place:

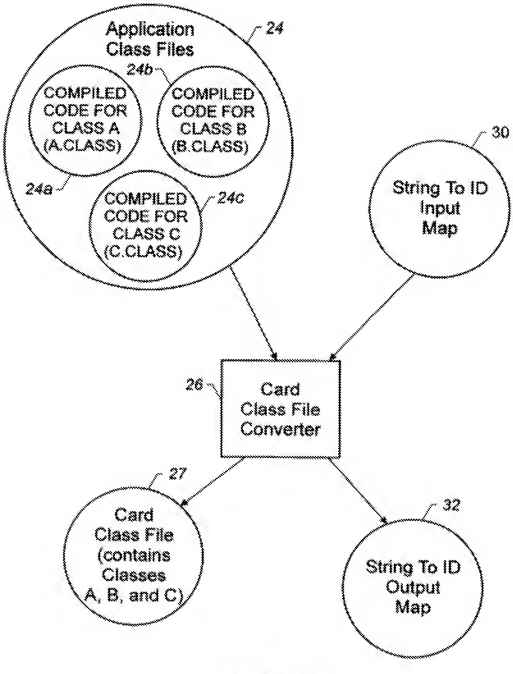


FIGURE 3

Figure 3 from '317

In the illustration of Figure 3, the compiled code for the class files are input to the card class converter 26. Also input is a String To ID Input Map 30. In the process of creating the card class file 27, the converter 26 replaces operand strings with unique identifiers. The converter 26 may also produce an output String to ID Map 32.

Appendix B of '317 (page 2) illustrates an example of the results of the operation. Consider the code segment:

```
public class HelloSmartCard {
  public byte aVariable;

  public static void main() {
    HelloSmartCard h = new HelloSmartCard();
    h.aVariable = (byte)13;
  }
}
```

Program: HelloSmartCard.java from Appendix B of '317

When converted, the converter builds an output map 32, for example, as shown below:

Offset (in Constant Pool)	String	ID	Mapped New/ Mapped/Old
00000A	"Code"	4000	New
000011	"SourceFile"	4001	New
00001E	"ConstantValue"	4002	New
00002E	"Exceptions"	4003	New
00003B	"HelloSmartCard"	F000	Old
00004C	"java/lang/Object"	F002	Old
000062	"<init>"	F004	Old
00006E	"()V"	4004	New
000074	"aVariable"	4005	New
00008A	"B"	FFF5	Old
00009E	"HelloSmartCard.java"	4006	New
0000B3	"main"	F001	Old

Relevant entries of String-ID OUTMap

Output Map from Appendix B of '317

The output map illustrates that, for example, the string "HelloSmartCard" has been replaced by the identifier F000. However, it should be noted, for the purpose of contrasting to the cited prior art, that that replacement is not an assignment of a value to the symbol HelloSmartCard. Rather it replaces one symbol, the symbol string "HelloSmartCard", with another, the much shorter F000. The IDs would then be provided values as appropriate. In the example code, HelloSmartCard is a class. In interpreting

the code the identifier F000 would be assigned a value appropriate to associate F000 with that class.

In the Office Action it is asserted that "a replacement of string operands to identifiers is well known in the prior art as a *macro*." (Office Action, Page 13, lines 9-10). However, in reality a *macro* is something very different from "modifying byte code operands from references using identifying strings to references using unique identifiers." As noted, in *Tanenbaum*, (Reference 11, Page 412, line 27), "a macro definition is a method for giving a name to a piece of text." That said, "after a macro has been defined, the *programmer* can write the macro name instead of the piece of a program." (*Tanenbaum*, Reference 11, Page 412, lines 27- 29, emphasis added). "*Macro expansion* occurs during the assembly process and not during execution of the program." (Ibid, at 414, lines 13-14, emphasis added). Thus, the notion of macros, if at all relevant, teaches away from Patentee's claim because in fact, macros are expanded from the macro name into a longer string. In one sense, that is the opposite of the result achieved through Patentee's claim limitation.

Thus, if one, for the sake of argument, were to consider the assembly process to be the counterpart to the converter step, the macro expansion would substitute the macro definition for the macro rather than substitute the unique identifier for the string that it stands for. Thus, if one were to attempt a comparison between macros and Patentee's claimed step, the macro definition and expansion process is opposite of what Patentee claims.

Furthermore, one must consider that macro definition is a step performed by the programmer and macro expansion is a step performed by the assembler. Patentee's steps of defining an identifier to stand for a string and to substitute such an identifier for a string operand are performed entirely by the converter.

It should also be noted that macros are used to expand into a sequence of instructions. Patentee claims the "modifying byte code *operands* from references using identifying strings to references using unique identifiers." The claimed substitution is for a byte code operand and not for a sequence of instructions.

Thus, unlike a macro, the identifier does not stand for a sequence of code that a programmer defines.

For any of these reasons, *Tanenbaum* and other references to *macros* do not teach or suggest, at least, the claimed limitation of "modifying byte code operands from references using identifying strings to references using unique identifiers".

The Examiner also asserted that the *UNIX Programmer's Manual*, in Sections A.OUT(5) and Section NM(1) (Reference 9), teaches that the "concept of replacing strings with integers in object files has long been known in computer science as name binding." (Office Action, Page 13, lines 14-15). Patentee disagrees.

Name binding is merely the association of values with identifiers. For example, in a symbol table, that association may be between a variable name and the address at which the value of that variable will be stored. Thus, a symbolic debugger can respond to a request for a value of a particular variable by translating the variable name into a memory address and retrieving the contents of that memory address.

Patentee claims something entirely different. It should be clear from the foregoing that "modifying byte code operands from references using identifying strings to references using unique identifiers", e.g., replacement of string entries in a constant pool as produced by a standard compiler with identifiers is not an assignment of a value to that string identifier and is therefore not the same as name bindings in UNIX. Accordingly, the *UNIX*

Programmer's Manual (Reference 9) and the UNIX operating system generally do not teach or suggest this element of Claim 78.

The Examiner also suggests that *Tanenbaum's* teaching of a symbol table as a list of Symbol-Value pairs is applicable to this claim element. For the reasons discussed above with respect to the UNIX Programmer's manual, a teaching of symbol tables does not teach or suggest the modification of references to strings into references to identifiers as claimed.

For the foregoing reasons, Patentee submits that the prior art, e.g., *Tanenbaum* or the *Unix Programmer's Manual*, do not teach or suggest "modifying byte code operands from references using identifying strings to references using unique identifiers."

Recording Jumps and Destinations and *Relinking Jumps*

Turning now to the elements *Recording Jumps and Destinations* and *Relinking Jumps*.

Claim 78 recites "recording all jumps and their destinations in the original byte codes." Claim 78 (and several of the other independent claims) have been amended herein to include the limitation "relinking jumps for which the destination address is affected by the conversion operation". That limitation was previously found in Claims 65 and 78.

As noted above, these two steps of recording jumps and destinations and relinking jumps are related. The former saves information used by the latter in recovering from the changes to the code that result from the conversion step.

The Examiner offers *Gosling '999* as the reference for teaching "recording all jumps and their destinations in the original byte codes" ("Prior Art *Gosling* teaches a system that 'sequentially processes all the instructions, and for each instruction that is a conditional or unconditional jump a

representation of the target location for the jump is stored in the directory portion of the stack snapshot storage structure.'" (Office Action, Page 10, lines 13-15). Patentee concedes that *Gosling '999* may teach that element. In the context of the converting step, the "recording all jumps and their destinations" is relatively uninteresting by itself. The step of "recording all jumps and their destinations in the original byte codes" establishes a basis for "relinking jumps for which the destination address is affected by the conversion operation" (Claim 78). If the conversion process has changed the length of certain instruction sequences, jumps would be affected. When that has occurred, the converter adjusts the destination addresses. *Gosling '999* does not teach or suggest any conversion step. Thus, *Gosling '999* certainly does not teach or suggest the corresponding "relinking" step.

It is difficult to understand the stated motivation to combine *Gosling* with *Peyret*. The Examiner states "[a]n obvious motivation exists since, as indicated above, the Java language was previously known as the 'Oak' language taught in prior art *Gosling*. (J. Gosling, IR '95, page 111) Hence, a skilled artisan tasked with implementing a Java (HLL) based smart card, and having access to the teachings of *Gosling '999*[we presume '999 is intended here], would have knowingly modified the teachings of *Peyret'884*, with the teachings of *Gosling '999* in order to convert from the class file format. (The conversion of Java class files necessitates the recording of jumps and their destinations.) This would obviously be necessary in order to maintain the intended portability of Java. (See: J. Gosling, IR '95, pp. 115 "Portability", for example)." (Office Action, Page 11, lines 8-15).

As noted above, *Peyret '884* does not teach or suggest a language that would require conversion. Linking is not conversion. It is difficult to understand why a person would modify the teachings of *Peyret '884* with the teachings of *Gosling '999* to convert from the class file format when *Peyret*

'884 does not teach or suggest use of a language requiring conversion. To take the additional step of stating that the person is tasked with modifying *Peyret '884* to use a high-level language that requires conversion and therefore would use *Gosling '999* to learn to record jumps would be an improper hindsight argument using the patent as a recipe for finding both the problem to be solved, the general solution, and specific steps of the solution.

Furthermore, the stated reason that "this would also obviously be necessary in order to maintain the intended portability of Java" (Office Action, Page 11, lines 13-14) is not relevant. The process of converting does not have anything to do with portability of Java. Java programs designed for JVM are portable between various JVM implementations. Similarly, converted class files are portable between smart cards implementing a standard card class file format. However, that portability does not derive from recording jumps.

Furthermore, *Gosling '999* does not deal with tools used in program building such as compilers, assemblers, linkers and loaders. Conversion is an added step in program building. On the other hand, *Gosling '999* teaches a methodology for program verification. *Gosling '999* states that "[the] byte code verifier 240 is an executable program which verifies operand data type compatibility and proper stack manipulations in a specified byte code (source) program 221 prior to the execution of the byte code program 221 by the processor 206 under the control of the byte code interpreter 242." (*Gosling '999,* Reference 2, Col. 4, lines 48-53). The motivation is to verify a correctness of a program prior to execution "for identifying prior to execution … any instruction sequence that attempts to process data or the wrong type for such a byte code or if the execution of any byte code instructions in the specified program would cause underflow or overflow of the operand stack,

and to prevent the use of such a program." (*Gosling '999*, Reference 2, Col. 2, lines 1-7). While program verification is a nice thing to have, it does not provide motivation for combining with techniques useful in reducing program size. Thus, a person studying *Peyret '884*, even if, for the sake of argument, that person was tasked with modifying *Peyret '884* for use with a general purpose high-level language such as Java, would not be motivated to look to *Gosling '999* to learn how to achieve that because *Gosling '999* does not teach or suggest that it deals with a related topic.

Accordingly, Patentee submits that a person would not be motivated to modify *Peyret '884* with the teachings of *Gosling '999*.

Turning now to *Relinking Jumps*. This limitation is discussed with respect to Claim 65 : "*Waite*,[8] at 339 ... teaches ... that the original destination address of each jump is first recorded, so that it can later be adjusted or re-linked to reflect the potential changes in the address space of the linked executable file, where such changes were caused by other steps in the conversion or linking process". (Office Action, Page 16, lines 11-15). Patentee disagrees.

Waite deals with the forward reference problem that occurs during the assembly of a program. (*Waite*, Reference 7, Page 339, lines 20-25). A forward reference is an operand that precedes the defining occurrence corresponding to that reference. To translate such an operand to a correct address in the assembly process, in a first pass, the operand locations and defining occurrences are saved and in a second pass (or through back chaining) the operands are replaced with the corresponding destination values.

[8] W.M. WAITE, Assembly and Linkage, in F.L. Bauer and J. Eickel, eds., Compiler Construction: An Advanced Course (2d ed. 1976).

However, while this is indeed a recording of jumps and destinations, it is not a "relinking jumps for which the destination address is affected by the conversion operation" because the forward reference problem and the assembly process do not affect the destination addresses and certainly not by a conversion operation. Accordingly, *Waite* does not teach or suggest "relinking jumps for which the destination address is affected by the conversion operation."

Conclusion with respect to rejection of the independent claims under 35 USC 103(a) over Peyret '884 in view of Gosling '999.

The above comparison of the prior art to the claims illustrates that even a combination of the cited prior art references would not yield the invention claimed in '317. *Peyret '884* does not include a converter. *Peyret '884* does not teach or suggest the use of a language that would require conversion. The *Peyret article* explicitly teaches away from using general purpose high-level languages for the purpose implementing interpreted languages for programming of smart cards. Furthermore, the references cited against the various conversion steps all fail to teach or suggest the conversion steps these references are cited against. For these reasons, at least, the independent claims are non-obvious over *Peyret '884, Gosling '999,* and the other known prior art taken singly or in any combination.

The Examiner argued that "a functionally equivalent conversion process would be inherent in the Peyret reference. This inherency is evidenced by any of the relevant art listed below [on page 8 of the Office Action] that teach the use of a linker in assembling compiled modules into a single executable program." (Office Action, Page 7 line 20 through Page 8 line 2).

However, as concluded both in the prosecution of '317 and in the decision by the Board of Appeals in the continuation application (10/037,390 now U.S. Patent 7,117,485), *Peyret '884* does not teach or suggest the conversion step, much less the specific conversion steps recited in the '317 patent. For a limitation to be considered inherent in a reference, that limitation must necessarily be present in the thing described in the reference. However, *Peyret's article*, Reference 13, teaches away from using existing high-level languages and teaches the use of a special purpose programming language that produces code tailored for the specific environment encountered in smart cards. Such a language would produce very small code. *Peyret article*, Reference 13, Page 63. Therefore, a conversion process would not be needed. Therefore, since *Peyret article*, Reference 13 teaches a viable alternative not requiring a conversion process, it follows that the conversion process is not necessarily present in the thing described in *Peyret '884*. Accordingly, a conversion process is not inherent in *Peyret '884*.

The detailed explanation of the conversion process provided hereinabove must illustrate that deriving the inherency thereof from the linker prior art is not appropriate because of the process of linking and the process of conversion have not much, if anything, in commmon.

Accordingly, for the reasons given above, the claims are not obvious.

Motivation to modify and motivation to combine

Patentee has discussed hereinabove in detail the references cited against the independent claims. As must be appreciated from the foregoing discussion of the prior art, even combinations of the references would fail to result in the claimed invention. It is, thus, not necessary to address the appropriateness of combining the cited references.

That said, Patentee contends that the combinations would not be appropriate even under the standards set forth in *KSR*:

> "Following these principles may be more difficult in other cases than it is here because the claimed subject matter may involve more than the simple substitution of one known element for another or the mere application of a known technique to a piece of prior art ready for the improvement. Often, it will be necessary for a court to look to interrelated teachings of multiple patents; the effects of demands known to the design community or present in the marketplace; and the background knowledge possessed by a person having ordinary skill in the art, all in order to determine whether there was an apparent reason to combine the known elements in the fashion claimed by the patent at issue." *KSR,* at 1741.

As noted above, the *Peyret article* teaches away from attempting to devise a system as the one presented in the '317 patent. *Peyret '884* does not teach a system in which conversion is required. Applicants posit that application of the various references cited to *Peyret '884*, while that would not arrive at the claimed solution, would require much more than a "simple substitution." When looking to the interrelated teachings of these various references (including the teaching away found in the *Peyret article*) and the background knowledge possessed by a person skilled in the art, it would be clear that making such a combination would not be apparent (besides not yielding the claimed invention). Accordingly, Patentee respectfully submit that creating these proposed combinations would not be obvious.

For this additional reason, the independent claims are not obvious over the cited prior art, including *Peyret '884* taken in any combination with the other cited prior art references.

Independent Claims 1, 31, 58, 64, 65, 84, 85, 86, 87

While the discussion above has been with reference to Claim 78 as a model, the other independent claims have analogous limitations to those set forth in Claim 78 and discussed hereinabove. Those arguments are therefore applicable to the other independent claims. Accordingly, Claims 1, 31, 58, 64, 65, 84, 85, 86, 87 are also valid and non-obvious over the cited prior art for, at least, the reasons given in support of Claim 78.

Dependent Claims

The dependent claims were rejected over combinations of *Peyret '884,* the other references discussed hereinabove, and the following references:

JEROME H. SALTZER and MICHAEL D. SCHROEDER, The Protection of Information in Computer Systems, in COMMUNICATIONS OF THE ACM, VOL. 17, NO. 7 (July 1974) (Reference 6)

ISO/IBC standard 7816-4 (1st ed. 1995) (Reference 12)

U.S. Patent 5,915,226 to Martineau (Reference 3)

These references also fail to teach or suggest a conversion process as set forth in the independent claims and do not teach or suggest the specific conversion steps recited in the independent claims. Accordingly, the independent claims are non-obvious over combinations that include these references.

The dependent claims incorporate all the limitations set forth in the independent claims, provide further unique and non-obvious combinations, and are therefore non-obvious over the prior art for the reasons given in

support of the independent claims and by virtue of such further combinations.

CONCLUSION

From the foregoing reasons, the independent claims in the present patent are valid and non-obvious over the cited prior art.

It is submitted that all of the claims are valid. Patentee respectfully request the issuance of a favorable reexamination certificate. If the Examiner believes that the prosecution of the application would be facilitated by a telephonic interview, Patentee invite the Examiner to contact the undersigned at the number given below.

Respectfully submitted,

Date: February 21, 2008

/Pehr Jansson/
Pehr Jansson

Registration No. 35,759

Pehr Jansson
The Jansson Firm
9501 N Capital of Texas HWY #202
Austin, TX 78759
512-372-8440
678-868-0101 (Fax)
pehr@thejanssonfirm.com

Attachments

Appendix A: Zhiquen Chen, Sun Microsystems, *Java Card Technology for Smart Cards, Architecture and Programmer's Guide*, page 7 (2000)

Java Card™ Technology for Smart Cards

Architecture and Programmer's Guide

Zhiqun Chen

ADDISON-WESLEY

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data
Chen, Zhiqun, 1969-
Java Card technology for smart cards : architecture and programmer's guide / Zhiqun Chen.
p. cm. — (The Java series)
Includes bibliographical references and index.
ISBN 0-201-70329-7 (alk. paper)
1. Java (Computer program language) 2. Smart cards. I. Title. II. Series.
QA76.73.J38 C478 2000
005.13'3—dc21 00-036360

The publisher offers discounts on this book when ordered in quantity for special sales. For more information, please contact:
Pearson Education Corporate Sales Division
One Lake Street
Reading, Massachusetts 01867
(800) 382-3419
corpsales@pearsontechgroup.com

Visit us on the Web at www.awl.com/cseng

ISBN: 0-201-70329-7
Text printed on recycled and acid-free paper.

1 2 3 4 5 6 7 8 9-CRS-04 03 02 01 00
First Printing, June 2000

tion. The digital certificate is issued by a certificate authority that testifies to the authenticity of a public key. Applications using smart cards for authentication include Web site access control, digital signing of e-mail messages, and secure on-line transactions. Many other Internet applications can be envisioned.

In a *closed environment*, such as a corporation or a university, multiapplication smart cards can provide physical entrance to buildings and computer facilities, grant levels of network access to internal Web sites and servers, store and process administration data, and enable various financial transactions (paying for meals, purchasing snacks at vending machines, ATM withdrawals and deposits, and so on).

As smart card technology gains wider acceptance, smart cards are finding their way into everyone's wallet.

1.2 Challenges in the Development of Smart Card Applications

Developing a smart card application traditionally has been a lengthy and difficult process. Although the cards are standardized in size, shape, and communication protocol, the inner workings differ widely from one manufacturer to another. Most smart card development tools are built by the smart card manufacturers using generic assembly language tools and dedicated hardware emulators obtained from silicon chip vendors. It has been virtually impossible for third parties to develop applications independently and sell them to issuers. Therefore, developing smart card applications has been limited to a group of highly skilled and specialized programmers who have intimate knowledge of the specific smart card hardware and software.

Because there are no standardized high-level application interfaces available in smart cards, application developers need to deal with very low-level communication protocols, memory management, and other minute details dictated by the specific hardware of the smart card. Most smart card applications in use today have been custom developed from the ground up, which is a time-consuming process; it usually takes a year or two for a product to go to the market. Upgrading software or moving applications to a different platform is particularly difficult or impossible.

Further, because smart card applications were developed to run on proprietary platforms, applications from different service providers cannot coexist and run on a single card. Lack of interoperability and limited card functions prevent a broader deployment of smart card applications.